**1) 25228 — Cryptocurrency address collection & categorization system**

**What it is (in one line)**

A system that automatically finds cryptocurrency wallet addresses published across the internet, stores them with context (where they were found, any names/phone numbers near them), groups related addresses and makes them searchable and exportable.

**Why it matters**

Criminals sometimes publish or use crypto addresses for illegal activities (scams, ransomware, dark-web sales). If investigators have a searchable, categorized database of suspicious addresses and the context they appeared in, it helps trace money flow and link activity to people or events.

**How it works — simple flow**

1. **Crawl / Scrape sources**: Visit webpages (news, forums, paste sites) and collect text that likely contains crypto addresses.
2. **Extract addresses & context**: Use pattern matching (regular expressions) to find Bitcoin/Ethereum/etc. addresses and capture the surrounding text (names, phone numbers).
3. **Normalize & validate**: Verify addresses (some blockchains have checksum rules) and tag currency type.
4. **Enrich using blockchain APIs**: Query public blockchain explorers (e.g., Etherscan) to get basic on-chain info (first seen, transaction counts).
5. **Cluster & categorize**: Group addresses that appear together or show transactional links; classify likely activity (scam, donation, ransomware) using simple rules or ML.
6. **Store & serve**: Save data in a database. Provide a dashboard for searching, filtering, visualizing clusters and exporting CSV/JSON.
7. **Automation**: Run scrapers regularly so the database stays up-to-date.

**Simple tech & tools (beginner-friendly)**

* **Language & web scraping**: Python
  + Scraping libs: requests, BeautifulSoup (static pages), Scrapy (scalable), Selenium (for JS-heavy pages)
  + Regex for address extraction
* **Data validation / enrichment**:
  + Use blockchain explorer APIs: Etherscan (Ethereum), Blockchair, Blockchain.info (Bitcoin)
* **Storage**:
  + Relational DB: PostgreSQL (for structured data)
  + Optionally graph DB: Neo4j (for address linkage)
* **Clustering & analytics**:
  + Python: scikit-learn (k-means, DBSCAN), networkx for graph analysis
* **Dashboard / UI**:
  + Web frontend: React or simple Flask app + Bootstrap
  + Charts: Chart.js or Plotly
* **Infrastructure**:
  + Docker for packaging
  + Scheduler: cron or Airflow for periodic scraping
* **Export**: CSV, JSON endpoints

**Minimal Viable Product (MVP)**

* Scraper for a few public sites (e.g., pastebin, 2 public forum pages)
* Address extraction and validation
* Simple dashboard showing list of addresses + source + timestamp
* CSV/JSON export

**Stretch features**

* Deep-web sources / Tor (very sensitive — legal/ethical caution)
* Automatic PII (name/phone/email) extraction using NER (spaCy)
* Graph visualization of address clusters and transaction flows
* Auto classification of activity type using ML

**Demo idea (what judges will see)**

* Run a scraper on a known sample site → show addresses discovered
* Click an address → show context (the scraped snippet), enrichment (last tx date), cluster members
* Export a selected set as CSV

**Main challenges & risks**

* **Legal/ethical**: Scraping some sites or deep web content may be illegal or risky. Always use public, allowed sources and respect robots.txt / terms of service.
* **Data quality**: Many false positives — need good validation.
* **Attribution limits**: Blockchain addresses don’t directly reveal who owns them.

**Who does what in a small team**

* Scraping & ingestion (1 person)
* Data processing & validation (1 person)
* Dashboard & frontend (1 person)
* Optional ML/graph analytics (same as data person or another)

**2) 25238 — Threat rules in ELK Stack for detecting Advanced Persistent Threats (APTs)**

**What it is (in one line)**

Create detection rules and dashboards in the ELK Stack (Elasticsearch, Logstash, Kibana) to identify patterns of long-running, sophisticated cyberattacks (APTs) using log data from endpoints, servers, and networks.

**Why it matters**

APTs are stealthy attacks that can persist undetected inside networks. Security teams need automated rules and alerts that can spot suspicious sequences (credential dumping, lateral movement, abnormal PowerShell use) before major damage occurs.

**How it works — simple flow**

1. **Set up ELK**: Components collect, process, index, and visualize logs.
   * **Beats** (Filebeat, Winlogbeat) collect logs from endpoints/servers and send to Logstash/Elasticsearch.
2. **Ingest logs**: Centralize logs (Syslog, Windows Event Logs, web server logs, firewall logs).
3. **Parse & enrich**: Use Logstash to parse fields, normalize timestamps, add tags (source IP, user).
4. **Develop detection rules**: Write queries/rules that look for suspicious events (e.g., many failed logins followed by a successful login from same host).
5. **Alert & dashboard**: Kibana shows dashboards and triggers alerts when rules match.
6. **Testing**: Generate synthetic APT-like logs (using public APT samples or scripts) to tune rules and reduce false positives.

**Simple tech & tools**

* **ELK Stack**:
  + **Elasticsearch** (index/search engine)
  + **Logstash** (ingestion & parsing pipelines)
  + **Kibana** (visualization & alerting)
  + **Beats** (Filebeat for Linux logs, Winlogbeat for Windows)
* **Log generators / APT samples**:
  + Atomic Red Team, Caldera, public APT repo samples (for synthetic logs)
* **Detection/analytics**:
  + Use Kibana alerting (Watcher) or ElastAlert (third-party) for advanced alerting
* **Optional extras**:
  + Suricata for IDS network alerts (feeds into ELK)
  + Sigma rules (a generic format for threat detection rules) which can be converted to Elasticsearch queries

**MVP**

* ELK instance up and running (dockerized or VM)
* Ingest sample logs (Linux syslog + Windows event logs)
* 5–10 well-crafted detection rules (e.g., suspicious PowerShell commands, lateral movement indicators)
* A Kibana dashboard showing rule hits and timelines

**Stretch features**

* Use machine learning jobs in Elastic (anomaly detection) for baseline deviations
* Integrate threat intel feeds (STIX/TAXII) to correlate observables
* Automated response playbooks (e.g., isolate host) via integrations

**Demo idea**

* Play a recorded APT sequence or run a synthetic attack in a contained lab.
* Show logs flowing into Kibana, the detection rule firing, and an alert dashboard that explains the triggered rule and suggested mitigation.

**Main challenges & risks**

* **Quality of synthetic logs**: Need realistic logs to tune rules.
* **False positives**: A big problem — tune thresholds carefully.
* **Scaling**: ELK can be resource intensive; for demo use small sample data.

**Who does what**

* ELK setup & ingestion pipelines (1 person)
* Rule development & threat hunting logic (1 person)
* Dashboard & demo orchestration (1 person)

**3) 25172 — Transformer-based end-to-end Web Application Firewall (WAF) pipeline**

**What it is (in one line)**

An AI-powered WAF that uses Transformer models (a modern type of AI that understands context in text) to learn what normal web requests look like and flag anomalous or malicious web requests (like SQL injection, XSS), instead of relying only on static rules.

**Why it matters**

Traditional WAFs rely on fixed rules and signatures and can miss new, cleverly-crafted attacks. A Transformer-based WAF can learn patterns and spot novel attacks (zero-day) by recognizing unusual request structures or payloads.

**How it works — simple flow**

1. **Collect access logs**: Capture HTTP requests from Apache/Nginx (both historical and live).
2. **Parse & normalize**: Extract fields — method, URL path, query params, headers, body. Replace dynamic parts (IDs, timestamps) with placeholders so the model focuses on structure.
3. **Tokenize & prepare input**: Convert requests into token sequences (like words in a sentence).
4. **Train transformer model**: Teach the model on *benign* (normal) traffic so it learns what normal looks like.
5. **Real-time inference**: Deploy the trained model as a service. For each incoming request, the WAF asks the model: “is this normal?” If not, flag/block.
6. **Non-blocking & concurrent processing**: Handle many requests at once without delaying the site (use async workers, batching, or a sidecar approach).
7. **Incremental updates**: Periodically fine-tune the model with new benign traffic so it adapts without retraining from scratch.

**Simple tech & tools**

* **Language & modeling**: Python
  + Transformers: HuggingFace Transformers library (e.g., DistilBERT, RoBERTa, or lightweight custom transformer)
  + Training frameworks: PyTorch or TensorFlow
* **Preprocessing**:
  + Python scripts for parsing logs and normalizing fields
  + Tokenizers (HuggingFace tokenizers)
* **Serving / inference**:
  + FastAPI / Flask to serve model predictions as an API
  + Use model optimization: convert to ONNX or TorchScript for faster inference
  + Use GPU if available; otherwise CPU-optimized small models
* **Integration with webserver**:
  + Sidecar microservice or reverse-proxy pattern (Nginx sidecar that forwards requests to the WAF API)
  + Or Nginx module / LUA script to call the model service (advanced)
* **Concurrency / scaling**:
  + Asynchronous worker pools (Celery, asyncio) or lightweight server (Uvicorn + FastAPI)
* **Data pipeline**:
  + Kafka for streaming logs (optional), or simple file tailing with tail -F
  + Database to store anomalies (Postgres, Elasticsearch)
* **Containerization**:
  + Docker for packaging components

**MVP (safe & achievable for hackathon)**

* Generate synthetic benign logs for a sample web app (the problem gives 3 sample apps).
* Parse logs, normalize requests, train a small transformer (DistilBERT or even a smaller custom transformer) on benign data offline.
* Build an inference API that labels requests as “normal” or “anomalous”.
* Demonstrate by sending a few injected malicious requests (curl) and showing the model flags them (batch/offline or near-real time).

**Stretch features**

* Real-time non-blocking integration with Nginx using a sidecar and async inference with latency < X ms
* Incremental fine-tuning pipeline (only new benign data)
* Explainability: show which tokens made a request anomalous (attention visualization)
* Rate limiting & automated blocking mechanisms

**Demo idea**

* Start sample web app, show benign requests being allowed.
* Send crafted malicious requests (like SQL injection string) → show model flags and logs recorded in dashboard.
* Show a simple metric: detection rate (on demo inputs) and latency for detection.

**Main challenges & risks**

* **Data**: Need good benign data to avoid high false positive rates.
* **Latency**: Transformers can be slow; must optimize or use small models.
* \*\* Explainability\*\*: AI flags without clear reason can be hard to justify in production.
* **Incremental learning**: Doing safe, incremental fine-tuning without catastrophic forgetting requires care.

**Who does what**

* Log parsing & dataset generation (1 person)
* Model training & optimization (1 person, ML-focused)
* Serving & webserver integration + demo orchestration (1 person)

**Quick comparison recap (so it’s fresh)**

* **Easiest to deliver a working demo quickly:** 25228 (Crypto collector) and 25238 (ELK APT rules) — both are engineering-focused and demo-friendly.
* **Most research/ML heavy and risky for hackathon:** 25172 (Transformer WAF) — impressive but needs ML expertise and careful scope reduction.
* **Most visual for judges:** ELK dashboards (25238) and Crypto graphs (25228). WAF is impressive technically but less flashy unless you prepare a tight demo.